API Interception via DLL Redirection

In Windows, all applications must communicate with the kernel through API functions; as such, these functions are critical to even the simplest Windows application. Thus, the ability to intercept, monitor, and modify a program's API calls, commonly called API hooking, effectively gives one full control over that process. This can be useful for a multitude of reasons, including debugging, reverse engineering, and hacking (in all interpretations of the word).

While there are several methods which can be used to achieve our goal, this tutorial will examine only DLL redirection. This approach was chosen for several reasons:

1. It is relatively simple to implement.
2. It allows us to view and modify parameters passed to an API function, change return values of that function, and run any other code we desire.
3. While most other methods require code to be injected into the target process or run from an external application, DLL redirection requires only write access to the target application's working directory.
4. We can intercept any API call without modifying the target (either on disk or in memory) or any system files.

Tools and Prerequisites

The following software will be used throughout this paper. You may of course use whatever utilities to which you are partial, however, bear in mind that their specific usage and implementation may vary:

- Visual C++ – Used to compile our DLL files.
- OllyDbg – Used to examine the target application and any external modules.
- DumpbinGUI – Used to obtain a list of functions exported by a target DLL.
- Linkout.pl – A perl script used to automate the majority of our leg work (requires ActivePerl).

It is assumed that the reader has a fairly solid grasp on Win32 programming in C/C++, assembly language, and usage of the above mentioned applications (minus the linkout script of course). A basic understanding of other methods used for API hooking is also helpful.

What is DLL Redirection?

Since an executable imports API functions from DLL files, DLL redirection allows us to tell a program that the DLLs it needs are located in a different directory than the originals; in this way we can create a DLL with the same name as the original, which exports the same function names as the original, but each function may contain whatever code we like. There are two ways to achieve DLL redirection; the first method is sometimes referred to as “dot local” redirection:

“Applications can depend on a specific version of a shared DLL and start to fail if another application is installed with a newer or older version of the same DLL. There are two ways to ensure that your application uses the correct DLL: DLL redirection and side-by-side components. Developers and administrators should use DLL redirection for existing applications, because it does not require any changes to the application. “

In other words, dot local DLL redirection affords developers the ability to force an application to use a different version of a particular DLL file than that used by the rest of the system. For example, if an application called oldapp.exe only worked with an outdated version of user32.dll, then instead of replacing the user32.dll file in the system32 directory (potentially causing many other applications to break), you could tell it to load the older version of user32.dll from the program's current directory by creating an appropriate dot local file. All other applications will still load the newer DLL from system32 and remain unaffected. All that is needed is to create a dot local file (which is simply an empty file whose name contains the name of the target application followed by a .local extension; in this case it would be oldapp.exe.local), and place it and the older version of user32.dll in the same directory as oldapp.exe.

However, there are a few limitations. Most notably, according to MSDN, certain DLL files (called 'Known DLLs') cannot be redirected in Windows XP (this restriction does not apply to Windows 2000). A list of all Known DLLs can be found in the *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs* key; included in the list of known DLLs are kernel32.dll, user32.dll and gdi32.dll. In my experience however, this is not true - it seems that under Windows XP, an application will either allow you to redirect any DLL, or none at all. As such, if targeting a program running on the Windows XP platform, dot local redirection is a unreliable method, and should be used only on Windows 2000 machines.

The second method, and the one which we will be using, uses manifest files to achieve the same result. Manifest files use the same naming convention as dot local files (i.e., oldapp.exe.manifest), but are not empty files. They must contain certain XML-formatted information in order to function properly, or else the target application will fail to load. In addition, manifest files are only supported on Windows XP and Vista; however, they are far more reliable than using dot local redirection, and allow us to redirect any DLL file. (NOTE: I have only tested this under Windows XP; it is possible that some restrictions/changes may be applied to Windows Vista).

How to Use DLL Redirection

It is fairly simple to use either of the above mentioned methods to redirect API imports to a DLL file of our choice, however, as we will see later, full implementation of DLL redirection is somewhat more complex. For now, we will focus on the basics: getting programs to load DLLs from the current working directory.

Programs only use DLL redirection when told to – luckily, telling them to do so is fairly simple. For dot local redirection, the creation of a file called program_name.exe.local will cause the application to look in the present working directory for DLL files before looking for them in the system folders. Very simple, but as previously noted, not very reliable on modern systems.

Manifest files are a bit more complex, as there is some necessary XML information that must be stored in the manifest file in order for it to function properly. Below is an example of a manifest file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
   version="6.0.0.0"
   processorArchitecture="x86"
   name="redirector"
   type="win32"
```

```
/>
<description>DLL Redirection</description>
<dependency>
   <dependentAssembly>
      <assemblyIdentity
         type="win32"
         name="Microsoft.Windows.Common-Controls"
         version="6.0.0.0"
         processorArchitecture="X86"
         publicKeyToken="6595b64144ccf1df"
         language="*"
      />
   </dependentAssembly>
</dependency>
<file
         name="user32.dll"
/>
</assembly>
```

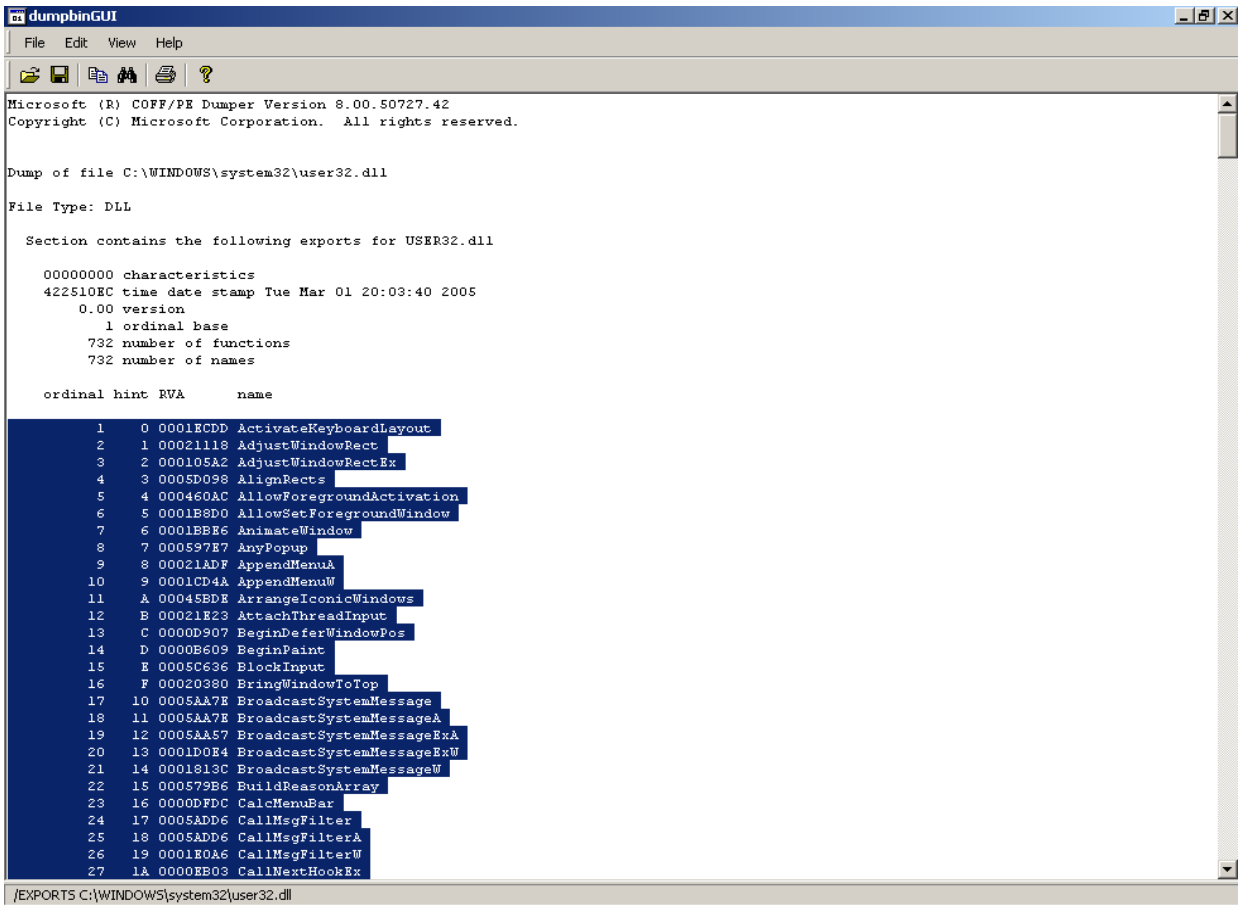
We will forgo a discussion on format of manifest files, as it is not directly related to the subject of this tutorial (more information can be found regarding manifest files here). However, note the <file> section; inside this section we have declared a name attribute, setting it equal to user32.dll. This tells the application to which the manifest file belongs that user32i.dll should be loaded from the current working directory. Once this file is created, save it using the same name convention used for dot local files (program_name.exe.manifest), and place it in the same directory as the target application.

Creating a Stub DLL

We now know how DLL redirection works, but like most things, it is a little bit more complex in practice: once we have redirected the program to load our DLL, we need to have all the functions it is looking for. Let's say that we want to intercept every call that Internet Explorer makes to MessageBox. MessageBox is located in user32.dll, so we will need to create a DLL called user32.dll that contains an exportable function named MessageBox, create the iexplore.exe.manifest file, and place our newly created user32.dll and iexplore.exe.manifest files in the C:\Program Files\Internet Explorer directory. Now when IE imports its API functions, it will load MessageBox from our user32.dll file; then, whenever IE calls the MessageBox function, the code we placed on our MessageBox function will be executed.

The catch is that MessageBox is not the only function that will be imported from user32.dll. There could be hundreds of functions that IE will be looking for in user32.dll and if any of these are missing then IE will fail to load. Since we don't want to re-write all of the user32 DLL functions, we will simply forward the rest of the functions to the original user32.dll file.

First, we use the dumpbinGUI utility to view all of the functions exported from user32.dll (right click user32.dll, and go to DumpbinGUI->EXPORTS). You should see something very similar to this:



Now, select the entire function listing as shown above (starting with ActivateKeyboardLayout, and ending with wvsprintfW), and copy it into a text file called user32.txt for later use. All of these functions will need to be exported from our DLL and forwarded to their corresponding functions in the original user32 DLL file. This can be accomplished using linker directives:

```
#pragma comment(linker, "/export:MessageBox=user33.MessageBox")
```

This instructs the linker to add an exportable function named MessageBox to our DLL's export table, and that this exported function will simply be a forwarder to the MessageBox function in user33.dll. Note the use of the name user33 instead of user32 (no, this is not a typo). This is necessary because our DLL is named user32.dll, so if we had specified user32.MessageBox, we would be forwarding the function back to ourselves. To prevent this, we will copy the original user32.dll file into the Internet Explorer directory (along with the manifest file and our new user32.dll file), renaming it to user33.dll.

User32 does not contain any, but some DLLs export functions by ordinal, and such functions will require some extra information. A function's ordinal number is the position in which it appears in the DLL file; in other words, a function with the ordinal 243 is the 243rd function exported by the DLL. In order for our DLL to export a function by ordinal and point it to the correct ordinal in the original DLL, we will use the following syntax:

```
#pragma comment(linker, "/export:ord243=shlwapi32.#243,@243,NONAME")
```

This tells the linker to export a function called ord243 with an ordinal value of 243, point it to ordinal

243 in shlwapi32, and do not include the name when exported. The only real difference here is the "@243" which instructs the linker to export this function with an ordinal value of 243, and the "NONAME" directive which tells the linker to not export this function by name. The name (ord243) is a random name; it does not matter what we put here since the function will not be exported by its name anyway.

To recap, we now have three files, all placed in the same directory as iexplore.exe:

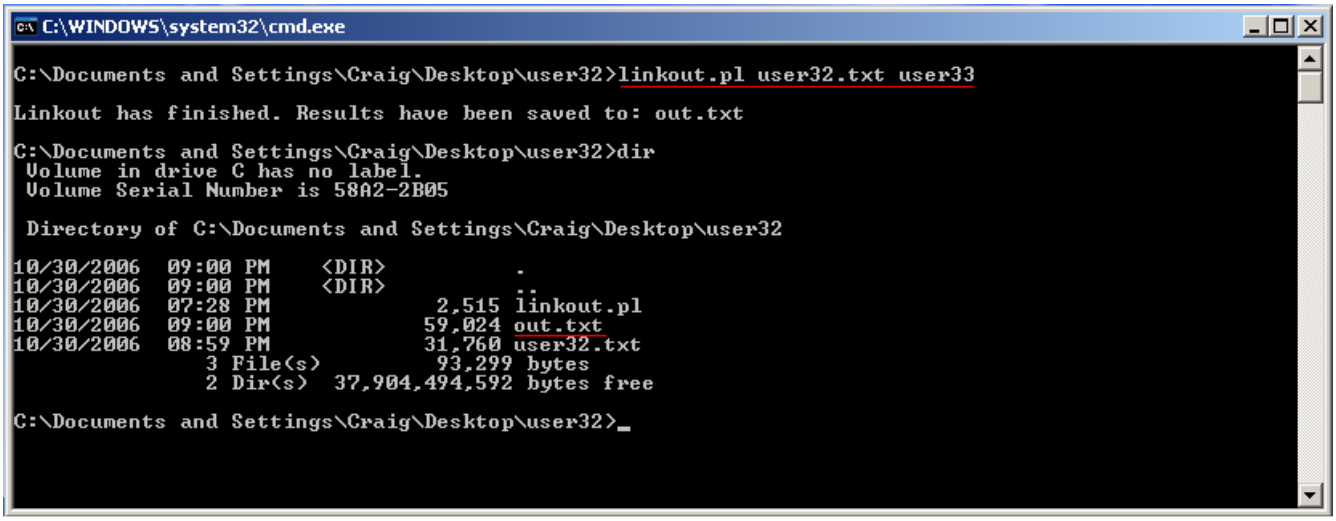
1. iexplore.exe.manifest – Indicates that user32.dll should be loaded from the current directory.

2. user32.dll – Our DLL that contains pointers to the original user32.dll file.

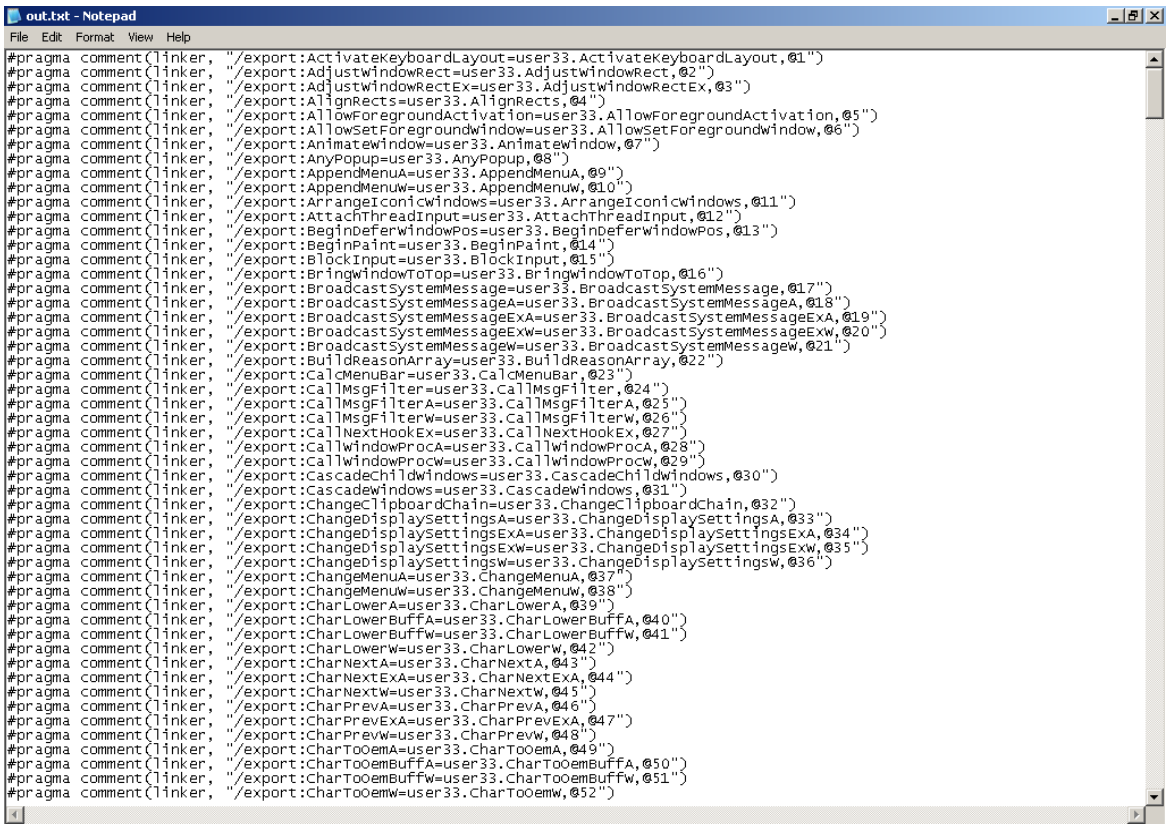
3. user33.dll – The original user32.dll file, which we have renamed.

Getting Our Hands Dirty

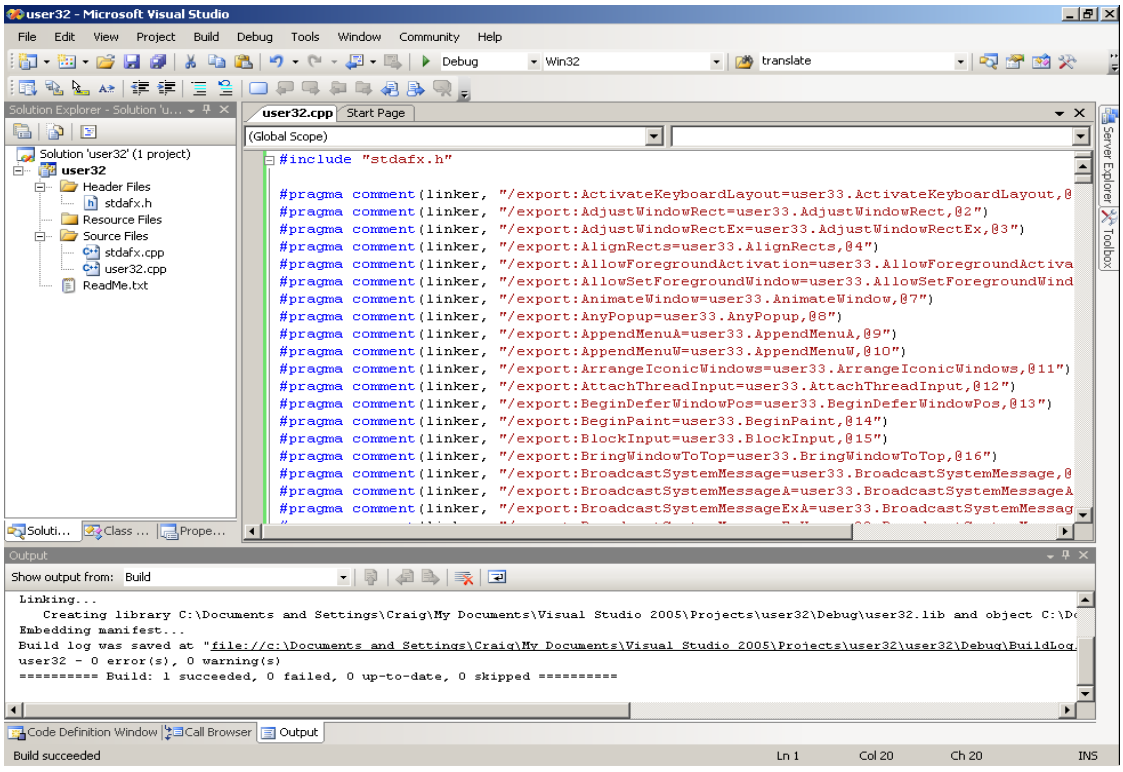
We have identified what needs to be done and how it can be done, now it's time to do it. There are a lot of functions in user32.dll, and manually creating a linker directive for each one would be time consuming. Instead, we are going to use a script to create the appropriate linker directives for each function using the user32.txt file created earlier, and the linkout.pl script. The linkout.pl script is pretty easy to use: just specify the name of the text file you saved the function list to (user32.txt), the name of the DLL you want the functions forwarded to (user33) and the output file you want to use (if none is specified, out.txt is used):



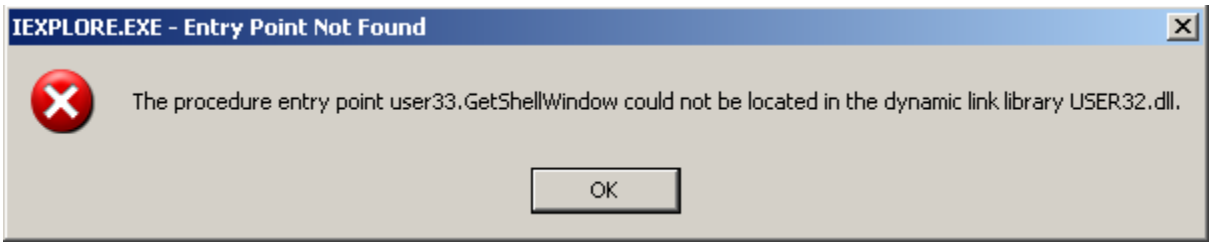
The out.txt file created by linkout.pl will look like this:



Now that all necessary linker directives have been generated, we need to copy them into the cpp file of a DLL project and compile it as user32.dll. Open up visual studio and create a new Win32 C++ DLL project called user32. You may delete all pre-generated code from user32.cpp (except for '#include “stdafx.h”") and paste the entire contents of out.txt into the file; then, build the project.



Copy the resulting user32.dll file into the Internet Explorer directory, as well as a copy of the original user32.dll file (remember to rename it to user33.dll). Finally, create the iexplore.exe.manifest file using the example provided earlier (note that IE does recognize dot local redirection, so you may create a dot local file instead if you desire). Start IE and it should run normally. To test that IE is in fact loading our DLL instead of the user32.dll located in the system32 folder, rename the user33.dll file to user34.dll and try running IE again. IE will fail to load with the following error:



This confirms that our DLL, whose GetShellWindow function points to user33.GetShellWindow, is being used.

Modifying Functions

So far our DLL really serves no purpose other than to forward functions to user33.dll. While this is a very necessary operation, the ultimate goal of all this is to modify some API calls. Let's look at what additional steps are necessary when intercepting and modifying and API call, using the Windows calculator as an example. Note that calculator does not recognize dot local redirection, so a manifest file must be used.

Create a copy of calc.exe and open it up in Olly. Since we already have a user32 stub DLL, let's look for calls to APIs located in user32.dll; the first one encountered is a call to GetProcessDefaultLayout:

Open up the user32 DLL project again, and comment out the linker directive for GetProcessDefaultLayout. Add the following code to user32.cpp:

```
__declspec ( naked ) void myGetProcessDefaultLayout(void)
{
	HINSTANCE handle;
	FARPROC function;
	DWORD retaddr;

	__asm{
		pop retaddr
	}

	handle = LoadLibraryA("user33.dll");
	if(!handle){
		MessageBoxA(NULL,"Failed to load user33.dll!","Error",MB_OK | MB_ICONERROR);
		ExitProcess(0);
	}

	function = GetProcAddress(handle,"GetProcessDefaultLayout");
	if(!function){
		MessageBoxA(NULL,"Failed to load GetProcessDefaultLayout!","Error",MB_OK | MB_ICONERROR);
		ExitProcess(0);
	}

	MessageBoxA(NULL,"GetProcessDefaultLayout called!","Hooked!",MB_OK);

	__asm{
		call far dword ptr function
		push retaddr
		retn
	}
}
```

In this function, we first pop the return address off the stack into the retaddr variable; we then find the address of the real GetProcessDefaultLayout using LoadLibrary and GetProcAddress. Next, we create a message box that will provide us with visual confirmation that we have successfully intercepted the API call. Finally, we call the real GetProcessDefaultLayout function ('call far dword ptr function'), push the return address back onto the stack, and return. Alternatively, we could have modified the parameters passed to GetProcessDefaultLayout, or changed its return value.

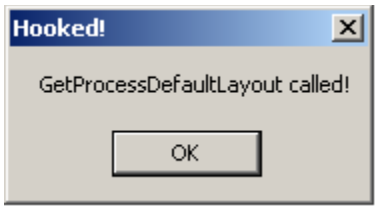
In order to properly export our new function, we will use a DEF file. DEF files can be used to create a list of whch functions should be exported, and what names they should be exported under. Create a file named user32.def in your project's folder and enter the following data:

```
LIBRARY	user32.dll

EXPORTS
	GetProcessDefaultLayout=myGetProcessDefaultLayout
```

This tells the linker to export the myGetProcessDefaultLayout function under the name of GetProcessDefaultLayout. Add the DEF file to your project (Project Properties -> Configuration Properties -> Linker -> Input -> Module Definition File)and compile the new user32 project. Copy/create user32.dll, user33.dll and calc.exe.manifest into the same folder as the copy of calc.exe.

Run calculator and you should be rewarded with the following message:



Combining DLL Redirection With Other Hooking Methods

DLL redirection can be very useful, however, attempting to intercept calls to functions located in critical DLLs such as user32 or kernel32 may result in application instability. Even though all functions are forwarded to their original functions, there may arise some cases in which this method results in unexpected and unwanted behavior. Other methods, IAT patching in particular, allow you to redirect just one function call, which minimizes this type of risk. By combining DLL redirection and IAT patching, you can achieve the best of both worlds: redirect only one function located in a critical DLL while still eliminating the need for an external program or DLL injection to patch the target application's IAT.

For example, you could find a function that is called early on in the program's execution, and located in a relatively obscure DLL, and perform DLL redirection to intercept this call. You could then implement the necessary code needed to patch the IAT entry for the target API before passing control back to the application.

Conclusion

DLL redirection can be a powerful tool for controlling user-land applications in Windows. It allows you to control any API function available for the Windows platform, thus allowing you to control existing program code (or to insert new code into the process) without modifying the application's code itself, either on disk or in memory. Such power also creates a great security threat, both to users and to software companies, as it could be used to compromise a user's system or to circumvent trial protection techniques (time-trials, CRCs, etc).